JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 223–243 (1999)

Research

Removing Clones from the Code

RICHARD FANTA and VÁCLAV RAJLICH*
Department of Computer Science, Wayne State University, Detroit MI 48202, U.S.A.

SUMMARY

This paper reports on a process for eliminating function clones and class clones from industrial object-oriented code. Clone removal can decrease system code size and facilitate maintenance. We eliminate clones by re-engineering scenarios that are based on using automated restructuring tools. The clone elimination process described includes function insertion, function expulsion, function encapsulation, function and variable renaming, and argument reordering. The paper presents examples of clones, re-engineering scenarios and restructuring tools. The usefulness of the approach is demonstrated by experience from a case study. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: function clones; class clones; clone removal; code restructuring; clone encapsulation; argument reordering

1. INTRODUCTION

Our research group has been studying the deterioration and restructuring of object-oriented industrial code (Fanta and Rajlich, 1998). During a review of industrial code we noticed that programmers often use identical or almost identical software components (e.g., classes and functions) in multiple places. These identical or almost identical components are called clones and previous research (Laguë *et al.*, 1997; Baxter *et al.*, 1998; Baker, 1995) suggests that clones account for 5%–15% of the code in large software projects. Section 6 of this paper surveys related work on clone removal.

One reason for the existence of clones in the program is that maintenance programmers do not fully understand the program and therefore they re-implement some already existing functionality. Another reason is time pressure and the effort not to introduce any bugs into already working code, particularly in a situation when the code contains complicated or not fully understood dependencies. In that case, programmers frequently copy-and-paste the code and update only some copies. Such duplications increase code size and lead to bloated code. They also make maintenance and comprehension more difficult, since they de-localize concepts implemented in the code. Because as much as 80% of the total life cycle cost is spent on maintenance, clone elimination could translate into substantial savings.

*Correspondence to: Dr. Václav Rajlich, Department of Computer Science, Wayne State University, Detroit MI 48202, U.S.A. Email: rajlich@cs.wayne.edu
Contract/grant sponsor: Ford Motor Co.
Contract/grant sponsor: National Science Foundation; Contract/grant number: 9803876.

CCC 1040–550X/99/040223–21$17.50
Copyright © 1999 John Wiley & Sons, Ltd.
Received 22 June 1998
Revised 25 May 1999

In order to remove duplicated code, the code must be restructured. The restructuring is performed in two phases:

- search for clones, and
- replace clones by a single code entity.

Other authors have investigated the identification of clones in code, as, for example, Laguë *et al.* (1997), Baxter *et al.* (1998), and Baker (1995). In this paper, we present the next step, which removes the previously identified clones. This paper deals with two kinds of clones: function clones and class clones. It discusses using restructuring tools in the broader context called re-engineering scenarios. This paper does not discuss clone detection as that builds on the results of previous detection research (Laguë *et al.*, 1997; Baxter *et al.*, 1998; Baker, 1995).

Clone removal requires substantial effort, because identical or near identical clones are often scattered throughout the code. When clones are removed, ripple effects can affect related code. The ripple effects are often hard to detect, particularly in large-scale projects with complex dependencies. Therefore, manual removal of clones is difficult and error prone.

In this paper we present a tool set that we used to remove clones in a medium-size C++ project. The tools analyse the code, make the desired changes and compensate for ripple effects. The tools also ensure that all preconditions hold before any transformation is applied. The individual tools presented in this paper perform small and efficient transformations. For more complex tasks, we use scenarios that combine the tools with human intervention.

In order to determine which tools need to be implemented, we adopt the following approach:

- search for clones in a medium size C++ program ('PET'),
- specify clone removal scenarios,
- implement transformation tools, and
- perform a case study.

Section 2 of this paper describes restructuring tools that we use for clone removal. Sections 3 and 4 define the use of the tools in re-engineering scenarios. Section 5 contains a case study reporting on applying the approach to industrial object-oriented code. Related work is reviewed in Section 6, and Section 7 contains conclusions.

2. TOOLS FOR RESTRUCTURING

2.1. Tool set overview

To support the restructuring of object-oriented code, we designed and implemented several high-level restructuring tools. The tools are universal and can be used in other projects as well. They cover editing changes that potentially affect large portions of code, i.e., transformations that have to be done simultaneously and consistently in several places. Transformations that affect only a few spots and produce no ripple effects are not covered and are performed manually with standard editors. The initial set contains:

- function insertion,
- function expulsion,

Copyright © 1999 John Wiley & Sons, Ltd.

```
class A {          int foo(B b,A& a){
public:            .
  int i;           a.i=4;
protected:         .
  char c;          .
};                 }
```

Figure 1. Class A *and function* foo() *before application of the insertion tool*

- function encapsulation,
- function and variable renaming, and
- argument reordering.

When using the transformation tools, the programmer specifies the input data and the tool performs the transformation on the entire code. After each transformation, the code can be successfully compiled and tested. Because of the complexity of C++, we decided to implement the transformations with certain limitations on their functionality. The limitations do not adversely affect common use of the transformations and they simplify tool implementation. The transformations and their limitations are summarized below.

2.2. Function insertion

Function insertion inserts a stand-alone function into a class as a new public member. Before the insertion, the target class must be one of the arguments of the function. An example of the starting situation is in Figure 1; the results are in Figures 2 and 3. The following code changes are performed:

- the transformation inserts the function header into the class specification;
- members of the target class are accessed directly in the function body, as exampled in Figure 2;
- the function header is qualified by the class identifier, and the parameter that is now replaced by membership is removed;
- all calls to the function are qualified with a class instance, as exampled in Figure 3; and
- all forward declarations of the function are removed.

In order to simplify the implementation we accepted the following limitations:

- the inserted function cannot be a member of any class;
- the inserted function cannot be called through a pointer, cannot be overloaded, cannot have a variable number of parameters, and cannot be a template function; and
- the function can be inserted only into a class of one of its parameters (if the parameter is a pointer to a class, then this pointer cannot be interpreted as an array or involved in any pointer arithmetic within the body of the function).

The function insertion tool takes the following data as input:

- name of the target class;
- position of the parameter that is to be replaced by membership;

Copyright © 1999 John Wiley & Sons, Ltd.

```
class A {                int A::foo(B b){
public:                  .
  int foo(B b);          .
  int i;                 this->i=4;
protected:               .
  char c;                .
};                       }
```

Figure 2. Code fragment after function `foo()` *was inserted into class* `A`

```
main() {                 main() {
A la;                    A la;
B lb;                    B lb;
...                      ...
foo(lb,la);              la.foo(lb);
...                      ...
}                        }
```

Figure 3. Call to function `foo()` *before and after its insertion into class* `A`

- name of the selected function; and
- list of files that are affected by the transformation.

After receiving its input data the tool analyses the source code and makes sure that all of the limitations hold. Then the tool performs the transformation. The tool accepts three different class parameter types to be converted to membership: reference, value and pointer. The tool also supports the insertion of recursive functions.

2.3. Function expulsion

Function expulsion is complementary to the function insertion. It removes a member function from a class and makes it a stand-alone function. The class is now passed to the function as a parameter. The following code changes are performed in the code:

- the function header is removed from the class specification;
- access to private and protected members is performed through public access functions;
- the class qualifier is removed from the function header and an additional parameter is added;
- members of the class accessed directly are accessed through the additional parameter;
- in all function calls, the qualifying instance is changed into a parameter; and
- a forward declaration of the expulsed function is added to the file that holds the source class specification.

In order to simplify the implementation, we accepted the following limitations:

- a pointer to the old function cannot be assigned to any pointer variable throughout the code;
- the old function cannot have a variable number of parameters;
- the old function cannot be a member of a class embedded into an inheritance hierarchy; and
- the old function cannot be a template or overloaded function.

Copyright © 1999 John Wiley & Sons, Ltd.

```
void foo(char c) {
int i,count,len;
char str[MAX];

cin>>str;
len=strlen(str);
count=0;
for(i=0;i<=len;i++)
  if(str[i]==c){
    count++;
    str[i]='\n';
    }
cout<<str<<" "<<count<<"\n";
}
```

Figure 4. Function `foo()` *with area selected for function encapsulation*

Even though this transformation is complementary to function insertion, its implementation is significantly more complex, because the function after expulsion cannot access private and protected members of the class. We deal with this problem by replacing direct access to such members with access through public access functions.

2.4. Function encapsulation

Function encapsulation packages a sequence of statements into a new function. The user selects a block of code for encapsulation and the tool decides whether the block is syntactically complete and can be changed into a function body. If yes, the tool creates a new function. The selected block is then encapsulated as a function body and replaced by a function call. If the encompassing function is a member of a class, the new function also becomes a member of the same class. In Figure 4, the framed text area represents code selected for encapsulation. Figure 5 shows the result of the encapsulation.

In the selected block of code, all variables are classified into one of the following four categories:

- local variable,
- global variable,
- parameter passed by value, and
- parameter passed by reference.

The tool generates a *local variable* if the original variable does not carry any data into or out of the selected block. Such a situation can occur even when the original variable is declared outside the selected code or when it is passed as a value parameter to the original function. The local variable must be written before being read inside the selected code, and if it is read in the outside code, it must be written in the outside code before it is read there.

If a variable is a *global variable* in the original code, it remains global in the encapsulated function. A variable will be passed as a *value parameter* to the new function if its value is used within the selected code but any modifications to this variable in this block are not used outside the block. It also must be declared local or passed by value in the original code. The rest of the variables

Copyright © 1999 John Wiley & Sons, Ltd.

```
void foo(char c) {
int i,count,len;
char str[MAX];

cin>>str;
len=strlen(str);
newfun(count,len,str,c);
cout<<str<<" "<<count<<"\n";
}

newfun(int& count,int len,
       char* str,char c){
int i;

count=0;
for(i=0;i<=len;i++)
    if(str[i]==c) {
        count++;
        str[i]='\n';
        }
}
```

Figure 5. Function `foo()` *after encapsulation of the* `newfun()` *function*

that do not qualify as local variables, global variables or value parameters are passed as *reference parameters*. In order to simplify the implementation, we allowed some input-only parameters to be classified as reference parameters. This, however, does not affect the code execution. Also, template functions are not supported. The transformation performed by this tool is similar to transformations proposed by Griswold (1991), Opdyke (1992), Lakhotia (1998), and Lakhotia and Deprez (1998).

2.5. Function and variable renaming

Renaming is used to avoid naming conflicts if a function or variable is moved between different name scopes. The tool is composed of two components: the code analyser and the text replacement tool. The code analyser analyses all of the identifiers in the given scope and makes sure that the new name does not conflict with any other identifier name (local, class member, global). In the second step, all old name occurrences are replaced by the new name.

2.6. Argument reordering

Argument reordering reorders function arguments while preserving code execution if the order of the arguments is significant. The tool makes the following changes:

- changes the order of the arguments in a function specification and in all forward declarations; and
- for function calls where the order of the arguments is significant, new auxiliary local variables are introduced, initialized in the original order and passed to the function in the new order.

This transformation cannot be used on overloaded functions. In some cases parameters are

Copyright © 1999 John Wiley & Sons, Ltd.

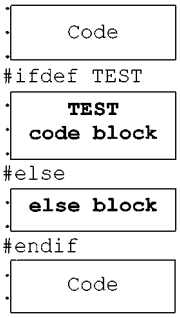

Figure 6. Code with conditionally compiled blocks defined using pre-compiler directives

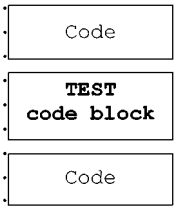

Figure 7. Code configuration after pre-compilation when variable TEST *was defined*

reordered using auxiliary variables even though their order is not significant. This inaccuracy, however, does not affect code execution.

2.7. Implementation issues

The transformations are implemented using GEN++ (Devanbu, 1992) for analysis of the source code and C++ for the actual changes to the code. GEN++ pre-compiles the code and then parses the result. The parsed code is stored in an abstract semantic graph and queried by the LISP-like scripting language that GEN++ provides. Several problems were encountered during the implementation of the transformation tools.

Pre-compiler directives, particularly conditionally compiled blocks, often occur in C++ code. Figure 7 shows the resulting code configuration when the variable TEST is defined in the example in Figure 6. Since the GEN++ parser pre-compiles before building the abstract semantic graph, code blocks that are excluded in the pre-compiler phase will not be parsed and thus, they will not be transformed. As a solution to this problem, we examine the code first in order to find all pre-compiler variables. Based on this pre-analysis, we run the GEN++ analyser several times to cover all possible software configurations, and the results of this multiple analysis are combined in a single output file.

The dependencies between C++ files may be changed by the transformations. As an example, consider the expulsion tool. Whenever a function is expulsed, its prototype is removed from the class definition. The C++ files that include the definition of that class lose the function prototype

Copyright © 1999 John Wiley & Sons, Ltd.

definition. This situation could cause compile time errors. As a solution, the expulsion tool adds an additional forward declaration of the function into the file that contains the class specification. In this way, the tools ensure that the code will compile successfully.

Complexities of C++ can make the implementation of the tools difficult. These complexities include automatic constructor and destructor invocations, automatic generation of temporary variables by the compiler, and pointer arithmetic. We deal with these problems by setting limitations for every transformation. The limitations concern unusual situations and did not occur in our case study.

The next two sections describe the use of the tools in restructuring scenarios. A restructuring scenario is a sequence of steps that are applied to a particular problem, where some steps are performed by the tools, while other steps are done by the programmer, who makes all key decisions.

3. FUNCTION CLONES

Semantically equivalent function clones are functions that provide identical functionality. They can have different names, different order and names of arguments, and different names of local variables. Programmers who do not fully understand the system introduce these clones by the re-implementation of an already existing functionality. These clones are different from function clones sharing common code, which are created when programmers copy-and-paste an already implemented function and modify one of the copies. Because of the different nature of the two kinds of the clones, different scenarios are used for clone removal.

Semantically equivalent function clones are replaced by a single function in the following scenario:

1. identify a group of semantically equivalent function clones;
2. select one function from the group;
3. reorder the argument lists of the clones to match the selected function's argument list;
4. replace all calls to the original clones with calls to the selected function; and
5. delete the clones.

In this scenario, step 1 is performed manually or with help of available clone detection technology (Laguë *et al.*, 1997; Baxter *et al.*, 1998; Baker, 1995). Step 2 is performed manually. Step 3 is performed using the argument reordering tool. Step 4 is performed using the automatic text replacement facilities available on ordinary editors. In step 5, no longer used clones are deleted by using the ordinary editor.

When dealing with clone functions sharing common code, we use the following scenario:

1. encapsulate common code into new functions (these functions are semantically equivalent clones), and
2. replace semantically equivalent function clones by a single function as in the previous scenario.

This scenario does not change the clients of the clone functions and therefore no change to the client code is necessary. An example of a scenario with three function clones is shown in Figures 8 to 12. Clones in Figures 8–10 update items stored in a linked list and share the code that searches the list. The shared code is encapsulated in the function `void search(Key,ListElement*&)`. The resulting code after the scenario is shown in Figures 11 and 12.

Copyright © 1999 John Wiley & Sons, Ltd.

```
void delete(Key key){
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found){
  if(eptr->data->key==key)
    found=true;
  else eptr=eptr->next;
  }
if(eptr){
  //code to delete
  //item from the list
  }
}
```

Figure 8. Code of `delete()` *clone function*

```
void update(Key key) {
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found){
  if(eptr->data->key==key)
    found=true;
  else eptr=eptr->next;
  }
if(eptr)
  eptr->data->trigger();
}
```

Figure 9. Code of `update()` *clone function*

```
Item* get_item(Key key){
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found){
  if(eptr->data->key==key)
    found=true;
  else eptr=eptr->next;
  }
if(eptr) return eptr->data;
else return NULL;
}
```

Figure 10. Code of `get_item()` *clone function*

Copyright © 1999 John Wiley & Sons, Ltd.

```
void delete(Key key) {
ListElement* eptr;

search(key,eptr);
if(eptr){
  // code to delete
  // item the list
  }
}

void search(Key key,
            ListElement* &eptr){
bool found;

eptr=head;
found=false;
while(eptr && !found){
  if(eptr->data->key==key)
    found=true;
  else eptr=eptr->next;
  }
}
```

Figure 11. Function `delete()` *after encapsulation of a new function* `search()`

```
void update(Key key){
ListElement* eptr;

search(key,eptr);
if(eptr)
  eptr->data->trigger();
}


Item* get_item(Key key){
ListElement* eptr;

search(key,eptr);
if(eptr)
  return eptr->data;
else return NULL;
}
```

Figure 12. Functions `update()` *and* `get_item()` *after clone removal*

4. CLASS CLONES

4.1. Almost identical concepts

Class clones are classes with identical or near identical code. Similarly to function clones, we classify them as class clones representing almost identical concepts, or class clones representing different concepts but sharing some code.

Copyright © 1999 John Wiley & Sons, Ltd.

```
class List1 {
public:
  void  Insert(Item*);
  void  StoreAtEnd(Item*);
  void  StoreAtBegining(Item*);
  bool  Delete(key);
  Item* Find(key);
  bool  Empty();
protected:
  eptr* head;
  eptr* tail;
};
```

Figure 13. Specification of class clone `List1`

```
class List2 {
public:
  void  InsertAtCurrent(Item*);
  Item* Current();
  bool  Remove();
  void  First();
  void  Advance();
  bool  Empty();
protected:
  eptr* head;
  eptr* tail;
  eptr* current;
};
```

Figure 14. Specification of class clone `List2`

Class clones representing almost identical concepts share implementation of function and data members. They are replaced by a single class, which contains a union of all class members. The following scenario describes how a single class replaces a group of class clones:

1. identify group of class clones;
2. select one class as the target class;
3. rename members of the other clones so that they have identifiers identical to the matching members of the target class;
4. copy data members from the other clones into the target class so that the data set is the superset of all of the clones;
5. expulse unique functions from the other clones;
6. replace all instances of the clones by the target class instances;
7. insert all the functions expulsed in step 5 into the target class; and
8. delete no longer used clones.

The rename tool supports step 3. The insertion and expulsion tools support steps 5 and 7. Step 6 is achieved by using the simple text replacement facility available in ordinary text editors. Unused clones are removed in step 8 by using ordinary editors.

We demonstrate the scenario in the example in Figures 13 to 17. The group of clones is represented by two classes `List1` and `List2` that both provide variations of list functionality, as shown in Figures 13 and 14. Both `List1` and `List2` share the implementation of function

Copyright © 1999 John Wiley & Sons, Ltd.

```
class List1 {
public:
  bool  Empty();
protected:
  eptr* head;
  eptr* tail;
};

void  Insert(Item*,List1);
void  StoreAtEnd(Item*,List1);
void  StoreAtBegining(Item*,List1);
bool  Delete(key,List1);
Item* Find(key,List1);
```

Figure 15. Class `List1` *specification after expulsion of unique functions*

```
void  Insert(Item*,List2);
void  StoreAtEnd(Item*,List2);
void  StoreAtBegining(Item*,List2);
bool  Delete(key,List2);
Item* Find(key,List2);
```

Figure 16. Headers of functions expulsed in step 5, after `List1` *was replaced by* `List2`

`Empty()` and the data members `head` and `tail`. Class `List2` provides functions for iterative access to its items while `List1` provides functions to support the ordered list. Both classes share implementation algorithms.

In step 2 of the scenario, `List2` is selected to replace `List1`, because its data members are a superset of the data members of `List1`. This simplifies the scenario because we can now avoid step 4. In step 5 all of the unique functions of class `List1` are expulsed using the expulsion tool, and the result is shown in Figure 15. In step 6 all of the occurrences of class `List1` are replaced by `List2` (see Figure 16). In step 7, all of the former functions of class `List1` are inserted into class `List2` using the function insertion tool. This completes the scenario and the final specification of class `List2` is shown in Figure 17. As the class `List1` is no longer used in the code it can be safely deleted. Figure 18 shows the original architecture, and Figure 19 shows the resulting architecture.

An advantage of this scenario is that it replaces a group of clones with a single class and thus simplifies the class dependencies. A disadvantage is that the scenario can create a class that does not represent a concept but is merely a coincidental union of the functionality of the original clones.

4.2. Separate concepts

Class clones representing separate concepts are classes that share some code, but it is not appropriate to replace them by a single class because such a class would not represent any concept. In this case the shared code is factored out into a new class. Each clone is then replaced by the composition of two classes, one of them being the new class containing all of the common code. The following scenario is used:

1. identify the common code for a group of clones;
2. rename the members so that matching members have identical identifiers;
3. for each clone, create a new class as a component of the original class;
4. move shared data members to the new component class;

Copyright © 1999 John Wiley & Sons, Ltd.

```
class List2 {
public:
  void  InsertAtCurrent(Item*);
  Item* Current();
  bool  Remove();
  void  First();
  void  Advance();
  bool  Empty();
  void  Insert(Item*);
  void  StoreAtEnd(Item*);
  void  StoreAtBegining(Item*);
  bool  Delete(key);
  Item* Find(key);
protected:
  eptr* head;
  eptr* tail;
  eptr* current;
};
```

Figure 17. Specification of class `List2` *after scenario application*

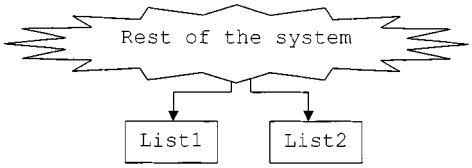

Figure 18. Original architecture contains two class clones `List1` *and* `List2`

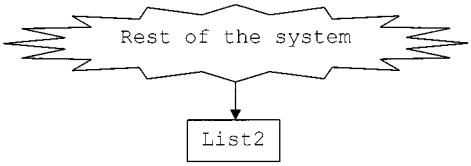

Figure 19. After scenario application, the system uses only class `List2`

5. move shared functions to the new component class; and
6. eliminate identical clones, as in the previous scenario.

Steps 4 and 5 are identical to scenarios for restructuring misplaced code, which we have described in detail (Fanta and Rajlich, 1998). Both scenarios used in steps 4 and 5 not only move data and functions into a component class but also apply compensating changes to all clients that were using moved members.

We demonstrate this scenario in the same example as the previous one. The classes `List1` and `List2` were selected in step 1. In step 2 two new classes were created: `BaseList1` and `BaseList2`. Figures 20 and 21 show the classes `List1` and `List2` and their new components after the application of steps 3 and 4. The final architecture after the application of step 5 is shown in Figure 22.

This scenario is more likely to create cohesive classes implementing specific concepts. A disadvantage is that the new classes it creates complicate the dependency graph. It also makes all

Copyright © 1999 John Wiley & Sons, Ltd.

```
class List1{
public:
  void       Insert(Item*);
  void       StoreAtEnd(Item*);
  void       StoreAtBegining(Item*);
  bool       Delete(key);
  Item*      Find(key);
  BaseList1* list;
};

class BaseList1{
public:
  bool       Empty();
  eptr*      head;
  eptr*      tail;
};
```

Figure 20. Specification of class `List1` *after extraction of a new class* `BaseList1`

```
class List2{
public:
  void       InsertAtCurrent(Item*);
  Item*      Current();
  bool       Remove();
  void       First();
  void       Advance();
  BaseList2* list;
protected:
  eptr*      current;
};

class BaseList2{
public:
  bool       Empty();
  eptr*      head;
  eptr*      tail;
};
```

Figure 21. Specification of class `List2` *after extraction of a new class* `BaseList2`

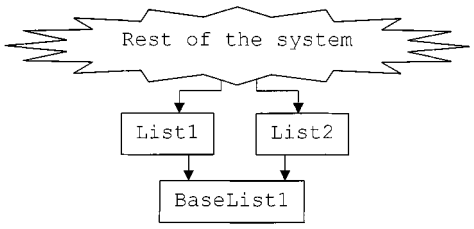

Figure 22. After scenario application, `List1` *and* `List2` *share common class* `BaseList1`

Copyright © 1999 John Wiley & Sons, Ltd.

of the moved data and functions public and therefore it violates the encapsulation rules. The proper encapsulation can be restored by additional transformations, which are not currently supported by our tool set. The scenarios presented in this section were applied to clones on the project 'PET' described in the next section.

5. CASE STUDY: PROJECT PET

PET (Powertrain Engineering Tool) is a CAD tool developed at Ford Motor Company (Mikulec, Li and Vrsek, 1996) to support the design of the mechanical components (transmission, engine, etc.) of an automobile. It is implemented in C++ and every mechanical component is modelled as a C++ class. Components are hierarchically composed into more complex components. For example, an engine is composed of an engine block, pistons, shafts, etc. Each component is characterized by a set of parameters dependent on the parameters of neighbouring components. The component dependency is described by a set of equations. Relationships among components and their parameters constitute a complex dependency network. Whenever a parameter value is changed, an inference algorithm traverses the entire network and recalculates the values of all dependent parameters. The value of each calculated parameter is checked for consistency against pre-set constraints. PET consists of 120 000 lines of C++ code, divided into 200 files and structured into approximately 80 classes and 50 global functions. It interfaces with other CAD software, including optimization software and 3D modelling software.

Ford engineers use PET to support transmission design. Therefore, all changes to PET were performed as quickly as possible in order to make the new functionality available. This situation prevented conceptual changes to the architecture, and the architecture progressively deteriorated. The introduction of the clones was one of the symptoms of this deterioration. During the code review we identified 10% of PET code as clones (both class and function clones). The longest clone found had 66 lines and the smallest clone recognized had 6 lines of code.

In the first phase of the case study, the code comprehension step was performed, because it is essential for clone detection. In Figures 23 and 24 we give examples of function clones found in PET. The code fragment shown in Figure 23 deletes the first 20 elements from the `return_str` array, where each array element represents a text line. This fragment repeats 14 times in PET. Another example of function clones in PET is the function `intersect` that returns the intersection of two linked lists. This function is defined in two clones in PET:

```
Linked_List* intersect(Linked_List*, Linked_List*)
Linked_List* LinkedList::intersect(Linked_list*)
```

A simple linear search algorithm is implemented in the fragment shown in Figure 24. The code fragment contains a linear search algorithm that finds an item placed in position `key` in the link list `apDtogglew_list`. There are eight function clones in PET, all implementing this algorithm. It will be used to illustrate how clones can affect maintenance.

The algorithm requires the items to be stored in a certain order in the list. If the code is changed so that the order is no longer guaranteed, the search algorithm must be also changed so that the item can be located on a different basis, as, for example, by its name. In that case, all eight clones must be visited and updated.

Copyright © 1999 John Wiley & Sons, Ltd.

```
//PET code
#ifndef DEBUG
cout<<"too many lines"<<endl;
#endif
int indx;
for(indx=0;indx<20;indx++){
  delete []return_str[indx];
  return_str[indx]=0;
  }
for(indx=20;indx<MAX_LINES;indx++){
  return_str[indx-20]=return_str[indx];
  }
for(indx=MAX_LINES-20;
    indx<MAX_LINES;indx++){
  return_str[indx]=0;
  }
numlines=MAX_LINES-20;
//PET code
```

Figure 23. The `too_many_lines` *clone example from PET code*

```
key = (int)pkey;
#ifdef DEBUG
cout <<"set_toggle_button"<<endl;
#endif
if(key == 0){
  XtVaSetValues(toggle_button,
                XmNset,True,NULL);
  }
else{
  if(apDtogglew_list != NULL) {
    tb=(Widget)apDtogglew_list->get_first();
    count=1;
    while(tb!=NULL && count!=key){
      tb=(Widget)apDtogglew_list->get_next();
      count++;
      }
    if(tb!=NULL){
      XtVaSetValues(tb,XmNset,True,NULL);
      }
    }
  }
```

Figure 24. The `set_toggle_button` *clone example from PET code*

In the second phase of the case study, the actual clone removal was performed in the 10 000 lines of code of the inference engine subsystem of PET. The clones were classified into different groups and the appropriate scenario was applied to each group. In the case of function clones we successfully removed exact duplicates and clones that perform identical operations on different data. However, we also located clones that contain common code but differ from each other by an embedded code sequence, expression or different data type. Such clones cannot be removed using our current tools, because the common code in these clones cannot be easily replaced by a single

Copyright © 1999 John Wiley & Sons, Ltd.

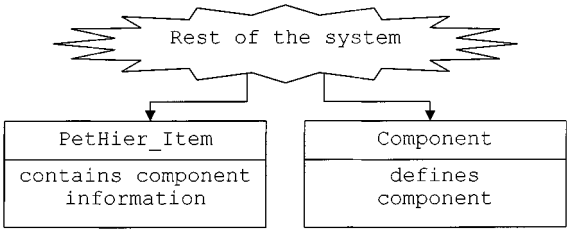

Figure 25. Original PET architecture uses class clones `PetHier_Item` *and* `Component`

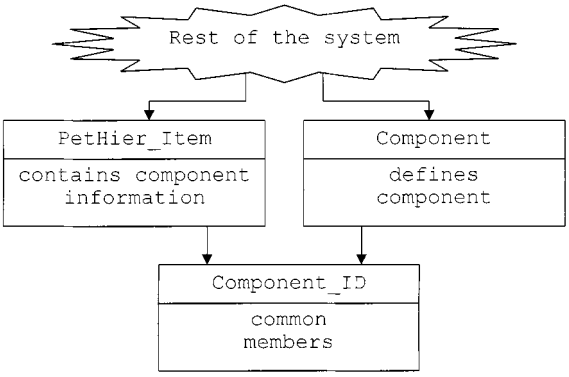

Figure 26. After restructuring, `PetHier_Item` *and* `Component` *both access* `Component_ID`

C++ function. A new and complicated set of tools would be required for the removal of these clones. We found that these clones constitute less than 3% of the examined code, and therefore we decided to leave the elimination of these clones to the programmer. This is in contrast to the research of Baxter *et al.* (1998) that replaces such clones by C++ macros and passes the different code parts as text replacement arguments. This approach works well, but we believe that the overuse of macros may produce hard-to-read and hard-to-analyse code that will create difficulties in future comprehension and maintenance.

In the case of class clones, we were able to merge all of the clones with identical functionality and extract the common parts of classes that share common code. Most of the class clones in PET are not exact duplicates and contain extra data or function members. An example is in Figure 25 where there are two classes describing a mechanical component, `PetHier_Item` and `Component`. Data contained in `PetHier_Item` are also present in the class `Component`. The same data are kept in two places and must be consistently updated and maintained. In order to remove the clone, we factored out the common data and members into a separate `Component_ID` class (see Figure 26).

6. RELATED WORK

Most of the related research is focused on clone detection and the assessment of the benefits of clone removal rather than on clone removal itself. The methods for clone detection are based on two principles: string-based (Baker, 1995) and AST-based—i.e., Abstract-Syntax-Tree-based (Baxter

Copyright © 1999 John Wiley & Sons, Ltd.

et al., 1998; Laguë *et al.*, 1997). Because of the differences in the definition of clones, these two groups report a slightly different percentage of clones in code.

Baker (1995) using the string based method detects exact and near-exact clones. This method uses a lexical analyser to detect clones that differ only by blanks and comments. The author reported that 13%–20% of the code in large-scale applications consists of clones.

Laguë *et al.* (1997) parsed code into AST form that is later converted into an intermediate representation language, and 21 different metrics are computed. The metrics characterize the functions in the following categories:

- name,
- layout,
- expressions, and
- control flow.

Based on the metrics, two kinds of clones are identified: ExactCopy and DistinctName clones. Using these methods, Laguë *et al.* (1997) found that clones represented 5%–10% of the code in a large software project. However, only exact duplicate functions and distinct name duplicate functions were considered.

The implications of duplicated code for software maintenance and reuse were investigated by Burd and Munro (1997). The study was conducted on legacy COBOL code. The results reported suggest that code duplication significantly increases code size, and it also increases de-localization of the same functionality throughout the code. In addition, Burd and Munro (1997) attributed numerous errors in the code to the cut-and-paste techniques that were responsible for the duplication of the code. Burd and Munro (1997) also investigated the relation of duplicated code and reuse. They argued that code replication is not reuse. Code replication (unlike reuse) increases cost and decreases the quality of the software. On the other hand, replicated code suggests a potential for reuse.

In parallel with our work, another interesting work (Baxter *et al.*, 1998) on clones was published. In this work, a parser processes the source code and produces an AST. In order to compare different sub-trees of the parsed code, a hash function is computed and only sub-trees hashed into the same bucket are compared. This method is able to identify a broad group of near-miss clones. Using the described clone detector, Baxter *et al.* (1998) reported a 13% clone content in a large software project. They also provided a simple clone removal tool that replaces clones with pre-processor macros.

The high-level editing tools for C++ code restructuring presented in this paper, are related to previous research (Griswold, 1991; Opdyke, 1992; Opdyke and Johnson, 1993a, b; Lakhotia, 1998; Lakhotia and Deprez, 1998). Opdyke and Johnson (1993a, b) studied the restructuring of classes related by composition and inheritance. Their transformation set included the creation of an abstract superclass, subclassing, and refactoring to capture aggregations and components. Among the transformations they proposed are transformations similar to our function encapsulation transformation (Opdyke, 1992). They also proposed a number of complementary low-level transformations. Refactorings proposed in their research were embedded into the Smalltalk Refactory Browser (Roberts, Brant and Johnson, 1997). The Refactory is an advanced browser that in addition to ordinary functions performs some of the refactorings. Among the features implemented in the Refactory is a drag-and-drop function that can be used to drop methods on classes or protocols.

Copyright © 1999 John Wiley & Sons, Ltd.

Griswold (1991) investigated meaning-preserving transformations in the block-structured language 'Scheme'. He defined meaning-preserving transformations on a program dependency graph that was extracted from the source code. Transformations were defined by graph transformation rules that manipulated statements within one block. The graph transformation rules included:

- transitivity—used to add a new variable to store an intermediate result,
- control—used to move an assignment statement into or out of a conditional block, and
- substitution—used to rename a local variable.

Lakhotia (1998) and Lakhotia and Deprez (1998) presented a collection of evolutionary transformations. The transformations are defined on a procedural language without global variables. The language contains an assignment statement, a branch statement and a function call statement where function call cannot be used in expression. The `fold` transformation is similar to our function encapsulation tool, as it creates a function from a set of statements. The transformation set also contains transformations that can bring together non-continuous regions of code.

Blaha and Premerlany (1996) give a catalog of transformations that can be used for object-oriented database restructuring. They use OMT notation (Rumbaugh, *et al.*, 1991) for transformation specifications, which makes the transformations independent of a specific programming language or database.

7. CONCLUSION

From our re-engineering experience on the PET system and from the findings of other research groups (Laguë *et al.*, 1997; Baxter *et al.*, 1998) we infer that clones in code are a serious problem, and that they are not considered as a type of reuse (Burd and Munro, 1997). Our experience shows that the negative factors of duplicated code include the following:

- clones add unnecessary code to the system, which must be maintained and understood;
- if a clone is changed it is likely that identical changes must be performed in the other clones as well;
- code polluted with clones has a complicated and confusing architecture; and
- cut-and-paste clone-creating techniques are error prone.

Data reported by Laguë *et al.* (1997) suggest that programmers in industry are aware of the problem and that they perform clone removal occasionally. However, we feel that specialized support for these activities is important. This support must include both clone detection and clone removal technology. The experience gained in the case study suggests that the advantage offered by the support increases with code size, because the length of the code that has to be scanned for potential ripple effects also increases.

However, the tools must be used by an experienced programmer who makes important decisions, as, for example, which functions are clones. The code after clone removal can still contain residual imperfections. In our future work we want to focus on the improvement of the analysers used for the restructuring, so that the tools could handle more complicated situations and produce better code. For frequently repeated actions we want to implement additional tools to reduce human intervention in the restructuring process.

Copyright © 1999 John Wiley & Sons, Ltd.

In our approach, the scenarios and tools for clone removal are very similar to the scenarios and tools for moving misplaced data and code (Fanta and Rajlich, 1998). When a new need for restructuring arises, the tools can be reconfigured into new scenarios. This gives us hope that a practical set of transformation tools for code restructuring can be gradually developed. This set would be suitable for all restructuring tasks in object-oriented code.

Acknowledgements

We acknowledge the support provided by Ford personnel, particularly Garry Vrsek, Antonín Mikulec, Libor Souček and Xianren Li.

References

Baker, B. (1995) 'On finding duplication and near-duplication in large software systems', in *Proceedings 2nd Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 86–95.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L. (1998) 'Clone detection using abstract trees', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 368–377.

Blaha, M. and Premerlany, W. (1996) 'A catalog of object model transformations', in *Proceedings 3rd Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 87–96.

Burd, E. and Munro, M. (1997) 'Investigating the maintenance implications of the replication of code', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 322–329.

Devanbu, P. (1992) 'GENOA—a customizable, language- and front-end independent code analyzer', in *Proceedings of the 14th International Conference on Software Engineering*, ACM Press, New York NY, pp. 307–317.

Fanta, R. and Rajlich, V. (1998) 'Reengineering object-oriented code', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 238–246.

Griswold, W. (1991) *Program restructuring as an aid in software maintenance*, Doctoral Dissertation, Department of Computer Science and Engineering, University of Washington, Seattle WA, 190 pp.

Laguë, B., Proulx, D., Mayrand, J., Merlo, E. and Hudepohl, J. (1997) 'Assessing the benefits of incorporating function clone detection in a development process', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 314–321.

Lakhotia, A. (1998) 'DIME: a direct manipulation environment for evolutionary development of software', in *Proceedings 6th International Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos CA, pp. 72–79.

Lakotia, A. and Deprez, J.-C. (1998) 'Restructuring programs by tucking statements into functions', *Information and Software Technology*, **40**(11–12), 677–689.

Mikulec, A., Li, X. and Vrsek, G. (1996) 'Transmission design with Powertrain Engineering Tool (PET)', in *Proceedings of the 9th International Pacific Conference on Automotive Engineering*, vol. 2, Ikatan Ahli Teknik Otomotif (IATO SAE), Jakarta, Indonesia, pp. 135–140.

Opdyke, W. (1992) *Refactoring object-oriented frameworks*, Doctoral Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign IL, 142 pp.

Opdyke, W. and Johnson, R. (1993a) 'Creating abstract superclasses by refactoring', in *Proceedings of the 21st Computer Science Conference*, ACM Press, New York NY, pp. 66–73.

Opdyke, W. and Johnson, R. (1993b) 'Refactoring and aggregation', in *Object Technologies for Advanced Software*, vol. 742 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 264–278.

Roberts, D., Brant, J. and Johnson, R. (1997) 'Refactoring tool for Smalltalk', *Theory and Practice of Object Systems*, **3**(4), 253–263.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs NJ, 500 pp.

Copyright © 1999 John Wiley & Sons, Ltd.

Authors' biographies:

Richard Fanta is a Ph.D. candidate in Computer Science at Wayne State University (WSU) in Detroit, MI, in the USA. His current research interests include object-oriented modelling and design, software maintenance, re-engineering and meaning-preserving software transformations. He received his M.S. degree from the Computer Science Department at the Czech Technical University (CTU) in 1993. Before coming to WSU he was a Ph.D. student in Computer Science and Engineering at the Czech Technical University and a Research Assistant at Charles University in Prague, Czech Republic. His email address is rif@cs.wayne.edu

Václav Rajlich is a Professor and former Chair of the Department of Computer Science at Wayne State University. Before that, he was an Associate Professor of Computer and Communication Science at University of Michigan in Ann Arbor, and software manager at the Research Institute for Mathematical Machines in Prague, Czech Republic. His current research interests include software maintenance, evolution and comprehension. He is a former General Chair of the IEEE International Workshop on Program Comprehension and the IEEE International Conference on Software Maintenance. His email address is: rajlich@cs.wayne.edu

Copyright © 1999 John Wiley & Sons, Ltd.